Computational Geometry: Theory and Applications 2 (1992) 93–110
Elsevier

COMGEO 121

Upper envelope onion peeling

John Hershberger
DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA

Communicated by Bernard Chazelle
Submitted 15 October 1990
Accepted 14 January 1992

Abstract

Hershberger, J., Upper envelope onion peeling, Computational Geometry: Theory and Applications 2 (1992) 93–110.

We consider the problem of finding the upper envelope layers of a set of line segments, sequentially and in parallel. The upper envelope of a set of n line segments in the plane can be computed in $O(n \log n)$ time (Hershberger, 1989). By repeatedly removing the segments that appear on the envelope and recomputing the envelope, one obtains a natural partition of the set of segments into *layers*. We given an $O(n \log n)$ sequential algorithm to find envelope layers if the segments are disjoint and an $O(n\alpha(n)\log^2 n)$ algorithm if the segments intersect ($\alpha(n)$ is the extremely slowly-growing inverse of Ackermann's function (Hart and Sharir, 1986)). Finally, we prove that the problem of finding envelope layers is P-complete, and hence likely to be intractable in parallel. The envelope layers problem is one of the first two computational geometry problems to be proved P-complete.

1. Introduction

Finding the convex layers of a point set (the onion-peeling problem) is a classical problem in computational geometry whose parallel complexity is still unresolved. Chazelle's $O(n \log n)$ sequential lagorithm [4] does not seem parallelizable, but neither has the problem been proven P-complete. On the other hand, layers of maxima can be computed quickly in parallel. Motivated by these two problems, we consider a related problem, the *envelope layers problem*.

The envelope layers of a set of line segments are analogous to the convex layers of a set of points, with convex hulls replaced by upper envelopes. The upper envelope of a set of (opaque) line segments in the plane is the collection of segment portions visible from the point $(0, +\infty)$. To define the envelope layers, we repeatedly compute the upper envelope of the set and discard the segments that appear on it (if any piece of a segment appears on the envelope, we discard the whole segment). The *envelope layers problem* is to label each segment with the iteration number at which it appears on the envelope.

The complexity of the upper envelope is the number of distinct pieces of segments that appear on it. In the worst case, the upper envelope of a set of n intersecting line segments may have complexity $\Theta(n\alpha(n))$, where $\alpha(n)$ is the functional inverse of Ackermann's function [9, 16]. The envelope can be constructed in optimal $O(n \log n)$ sequential time [10], and in $O(\log n)$ time on an $O(n)$-processor PRAM (Goodrich, personal communication).

If the segments are non-intersecting, then the complexity of their upper envelope is linear in n. Constructing the envelope takes $\Theta(n \log n)$ time. However, if the left-to-right order of the segment endpoints is given, then the upper envelope can be computed in $O(n)$ time [2].

This paper gives two algorithms for the envelope layers problem, a straightforward one for non-intersecting segments and a more complicated one for intersecting segments. The algorithm for non-intersecting segments exploits the fact that disjoint segments can be topologically ordered from top to bottom; it runs in optimal $O(n \log n)$ time and linear space. A similar algorithm was developed simultaneously by Overmars (personal communication). Intersecting segments cannot be ordered, and so the $O(n \log n)$ algorithm is inapplicable to them. The algorithm for this case is based on an output-sensitive method for computing the upper envelope in $O(\log^2 n)$ time per edge of the envelope, with an $O(n \log n)$ overhead; the bound for computing the envelope layers is $O(n\alpha(n)\log^2 n)$.

The second part of the paper addresses the parallel complexity of the envelope layers problem. We show that the problem is complete for P under NC reductions, and hence is probably intractable in parallel. (A P-complete problem is as hard as the hardest problems solvable in deterministic polynomial time; unless P = NC, such problems are not solvable in parallel using polylogarithmic time and a polynomial number of processors [7, 12, 14].)

The P-completeness of the envelope layers problem is important for two reasons. First, it is a step toward resolving the complexity of the convex layers problem. Second, the envelope layers problem is one of the first two geometric problems to be proven P-complete. (At about the same time as this research was performed, Atallah, Callahan, and Goodrich independently proved the P-completeness of the envelope layers problem, using a reduction different from ours, and a lexicographic planar partition problem [3].)

2. A sequential algorithm for disjoint segments

We first consider the case of a set S of disjoint segments, with $|S| = n$. This case is easier than the general case for two reasons. First, the segments can be ordered topologically from top to bottom. Second, the vertical lines through the segment endpoints partition the plane into *slabs* in which any vertical line intersects the

same segments in the same order. Both of these properties contribute to give an $O(n \log n)$ algorithm.

We can define a relation $\succ$, pronounced 'above,' on the segments of S. We say that $a \succ b$ if segments a and b are intersected by a common vertical line, and the intersection with a is above that with b. Because a and b do not intersect, we cannot have $b \succ a$; that is, the relation is antisymmetric. Guibas and Yao have shown that the 'above' relation is a partial order, and hence can be extended to a total order [8]. This total order can be found by computing the vertical visibility graph of the segments, which has linear size, and applying topological sort to it. The key property of the 'above' order is that the layer to which a segment belongs is affected only by segments 'above' it, as is easy to see by induction.

Partitioning the plane into slabs lets us rephrase the dynamic definition of layer depth in a static form. Consider a vertical ray extending upward from a segment s at x-coordinate $\hat{x}$; let $f(s, \hat{x})$ be the depth of the deepest segment encountered by the ray. Then the depth of the segment s is one greater than the minimum of $f(s, \hat{x})$, taken over all $\hat{x}$ between the endpoints of s. In each slab formed by drawing vertical lines through all the segment endpoints, the ray defining $f(s, \hat{x})$ intersects the same segments in the same order. Thus the problem of minimizing $f(s, \hat{x})$ is discrete rather than continuous: we minimize $f(s, \hat{x})$ over the $O(n)$ slabs that intersect s.

The previous two observations suggest a straightforward $O(n^2)$ algorithm. The algorithm inserts the segments in 'above' order from top to bottom; for each slab the algorithm maintains the maximum depth of the segments that span it. The slab depths are initially zero. When the algorithm inserts a new segment, it sets its depth to one greater than the minimum slab depth of the slabs that intersect the segment. It then updates the depth of each of those slabs to the maximum of its old depth and the depth of the new segment. (This raises the depth of some slabs by one and leaves the others unaffected.) It is straightforward to show by induction that this algorithm correctly computes the depth of each segment.

We can implement this algorithm to run in $O(n \log n)$ time using a segment tree. A segment tree is a complete binary tree whose leaves represent the slabs, taken in left-to-right order, and whose internal nodes represent the union of their descendants' slabs [15, pp. 13–15]. Each region associated with a node, whether an original slab or a union of them, is a *canonical slab*. Any segment can be decomposed into $O(\log n)$ subsegments, each with its endpoints on the boundary of some canonical slab. The unique decomposition of a segment into a minimum number of such subsegments is the *canonical decomposition* of the segment. We modify the quadratic algorithm described above to maintain, for each canonical slab, the minimum slab depth of the original slabs contained in it. To find the depth of a new segment, we compute the minimum slab depth over $O(\log n)$ canonical slabs, rather than over $O(n)$ original slabs as above.

We cannot afford to store the minimum depth of each canonical slab at the node representing the slab; to do so would force us to spend $\Theta(n)$ time updating

these values when long segments are added. Instead, we store at each node u a value $\text{val}(u)$ that satisfies the following invariant.

> For each canonical slab (represented by a node v), the minimum slab depth is given by the maximum of $\text{val}(u)$ over all ancestors u of v, including v itself.

We initialize $\text{val}(v)$ to zero for all nodes v, establishing the invariant at the beginning of the algorithm.

The canonical decomposition of a segment uses $O(\log n)$ canonical slabs, and hence $O(\log n)$ nodes of the segment tree, which we call *canonical nodes* for the segment. The parents of these nodes lie on two root-to-leaf paths in the tree. By walking down these paths, we can compute the minimum slab depth of all the canonical nodes, and hence the depth of the new segment, in $O(\log n)$ total time.

Once we have found the depth d of a new segment, we must update the tree to maintain the invariant. There are two parts to this task. First, we must ensure that each canonical node and its descendants have minimum depth at least d. To do this, we set $\text{val}(v)$ to the maximum of its old value and d, for each canonical node v. Second, we must maintain the invariant for ancestors of the canonical nodes: increasing the minimum slab depth of one child of an ancestor may increase the minimum slab depth of the parent. We walk through the $O(\log n)$ ancestors of canonical nodes from bottom to top. At each ancestor v with children l and r, we set $\text{val}(v)$ to $\max(\text{val}(v), \min(\text{val}(l), \text{val}(r)))$. Both updates take $O(\log n)$ time altogether. The following theorem proves the correctness of this algorithm.

Theorem 2.1. *The segment tree algorithm finds the depth of each segment correctly and runs in* $O(n \log n)$ *time and linear space.*

Proof. The previous paragraphs show that the algorithm runs in $O(n \log n)$ time: $O(n \log n)$ for the initial computation of the 'above' order, and $O(\log n)$ for each of n segment insertions. The invariant means that the depth of each segment is correctly computed; we must show that the invariant is maintained throughout the algorithm.

The first step of invariant restoration sets $\text{val}(v) := \max(\text{val}(v), d)$ for every canonical node v. This restores the invariant for v and all its descendants: the maximum val field on the path from the root to the node is at least d, and no smaller than it was before. The second step sets $\text{val}(v) := \max(\text{val}(v), \min(\text{val}(l), \text{val}(r)))$ for each ancestor v of the canonical nodes, working from the bottom up. We prove by induction that this operation immediately restores the invariant for each node to which it is applied, and does not destroy it for any node.

The basis of the induction is established for the canonical nodes by the first step. When we apply the operation to a node v, the invariant may or may not

hold for v already. If it does, the operation does not change that, since val(v) does not decrease, and does not increase without justification (the minimum slab depth of v is equal to the minimum slab depth of its two children, which is larger than the minimum of the val fields of the children). If the invariant does not hold for v already, then the minimum slab depth of the children of v (which is the depth of v) is greater than val(u) for any ancestor u of v, including v itself. Because the invariant holds for the children of v, the lesser of their val fields must be equal to the depth of v. The operation sets val(v) to this value, restoring the invariant for v. Changing val(v) affects the invariant only for v and its descendants. Because the depth of any node is at least as great as the depth of its ancestors, restoring the invariant for v does not violate it for any of its descendants.

To complete the proof of correctness, we note that the insertion of the new segment does not affect the minimum slab depth of nodes that are neither descendants nor ancestors of canonical nodes (call them *noncombatants*). Thus the invariant holds for noncombatants before any invariant restoration has been done. Neither restoration step affects the noncombatants: the first step does not affect their ancestors at all, and the second step does not change any ancestor's val field without justification. If an ancestor's val field changes, then the depth of the noncombatant must already have been at least as great as the new value. □

3. A sequential algorithm for intersecting segments

We now consider the case of intersecting line segments. For convenience, we assume that all intersections are proper—no collinear segments overlap, and no segment endpoint lies in the interior of another segment—but we do allow segments to share endpoints. Because intersecting segments do not have an 'above' ordering, the algorithm of the previous section is inapplicable.

The algorithm for intersecting segments closely follows the definition of envelope layers. It repeatedly computes the upper envelope, then deletes the segments on it. If the algorithm were to build the upper envelope from scratch at each iteration, it could take $\Theta(n^2 \log n)$ time to find the layers. This section shows how to improve on this bound by using an output-sensitive algorithm to compute the upper envelope. The algorithm is too costly to use to compute a single upper envelope—it has an overhead of $\Omega(n \log n)$ time and space, and is more complicated than the known $O(n \log n)$ algorithm [10]—but it is well suited to layer computation. The algorithm produces envelope edges at a cost of $O(\log^2 n)$ apiece, but more importantly, its data structures allow segments to be deleted at an amortized cost of $O(\log^2 n)$ apiece. This means that the cost of finding the layers is $O(n \log^2 n)$ for deletions, plus the total cost of building envelopes. If n_i segments appear in the ith layer, the worst-case size of the ith envelope is

$\Theta(n_i\alpha(n_i))$. To find all the envelopes, the algorithm uses at most

$$O\left(\sum_i n_i\alpha(n_i)\log^2 n\right) = O(n\alpha(n)\log^2 n)$$

time and $O(n \log n)$ space.

The output-sensitive algorithm first finds the leftmost segment of the upper envelope, then traces along the envelope. For each envelope segment, it finds where the segment leaves the envelope, computes its successor, and repeats the process until it reaches the right end of the envelope. The key to the algorithm is finding where the current segment leaves the envelope. The data structures used to solve this problem also support finding the leftmost envelope segment to the right of a given x-coordinate, an operation needed, for example, at the beginning of the algorithm.

Finding the end of the current envelope segment is a kind of 'shooting' problem [1, 5]. The problem specifies a collection of segments S, a particular segment $e \in S$, and a point $p \in e$ on the envelope of S. Let $\vec{e}$ be the ray with origin p that extends e to the right. The shooting problem asks for the first place where $\vec{e}$ leaves the upper envelope of $\vec{e} \cup S$. If this point is to the right of the right endpoint of e, then e leaves the envelope at its right end; otherwise $\vec{e}$ and e leave the envelope together.

To make the shooting problem easier, we regularize the segments of S. As noted in Section 2, each segment of S can be broken into $O(\log n)$ canonical subsegments based on a segment tree for S. Each canonical subsegment (call it a *fragment*) is associated with some node of the segment tree: the segment's endpoints lie on the boundary lines of the node's slab. The shooting data structure places the canonical fragments into $O(\log n)$ groups, based on the levels of their associated segment tree nodes. Canonical fragments associated with level i of the segment tree belong to group S_i (the root is level 0).

Each group S_i has a corresponding partition of the plane into at most 2^i-vertical slabs. Each segment fragment in S_i belongs to exactly one slab, and extends all the way across it. Thus within a slab, the fragments look like infinite lines. The upper envelope of infinite lines is a single convex face. The upper envelope of a group S_i, therefore, is a sequence of piecewise linear convex functions, one per slab, which we call *cups*.

The shooting algorithm finds the place where $\vec{e}$ leaves the envelope of $\vec{e} \cup S_i$ for each group S_i. Because $S = \bigcup_i S_i$ and the shooting problem is decomposable, the leftmost such point is the place where $\vec{e}$ leaves the envelope of $\vec{e} \cup S$. Each group S_i is a sequence of cups. Shooting into a cup is simplified by considering the *cup endpoints* (the intersections of the cup with its slab's boundary). For purposes of shooting, a cup's endpoints characterize the cup (see Fig. 1).

Lemma 3.1. *The ray $\vec{e}$ leaves the envelope of $\vec{e} \cup S_i$ in the leftmost cup of S_i for which $\vec{e}$ passes strictly below a cup endpoint.*

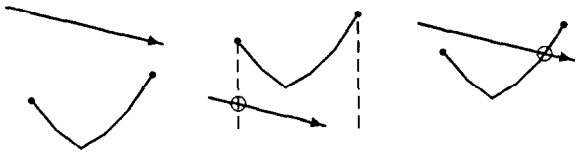


Fig. 1. Configurations of $\vec{e}$ and a cup.

Proof. Refer to Fig. 1. By convexity, $\vec{e}$ cannot leave the upper envelope in a cup without passing beneath one of its endpoints. The restriction that intersections be proper means that any endpoints that $\vec{e}$ passes through must belong to fragments of e. □

Once we have found the cup of S_i in which $\vec{e}$ leaves the envelope of $\vec{e} \cup S_i$, completing the shooting query is not difficult, given the right supporting data structures. If $\vec{e}$ passes below the left endpoint of the cup, then $\vec{e}$ leaves the envelope there. Otherwise, $\vec{e}$ leaves the envelope at an intersection with the cup inside the slab. Binary search on the cup's segments can find the intersection in $O(\log n)$ time.

Before considering the problem of determining which cups to shoot into, let us discuss the representation of the cups. There are $O(n)$ cups, each one associated with a segment tree node, and $O(n \log n)$ canonical segment fragments distributed among them. Each cup is the upper envelope of fragments in a slab, but because the fragments span the slab, it can be regarded as the upper face of an arrangement of lines, clipped to the slab. Because the envelope layers algorithm deletes segments, we need a data structure that represents the upper face of an arrangement of lines, allows deletions from the set of lines, and supports binary search on the boundary of the upper face.

By applying geometric duality [6], we can solve the problem using the *hull tree* data structure of Hershberger and Suri [11]. Duality maps lines to points and maps the upper face of the line arrangement to the convex hull of the points. The problem of intersecting the face with a line maps to the problem of finding the line(s) supporting the convex hull and passing through a query point. The hull tree is ideal for this dual problem: the convex hull of k points can be built in $O(k \log k)$ time and $O(k)$ space; deleting a point takes $O(\log k)$ amortized time; and finding a supporting line through a query point takes $O(\log k)$ time. Because there are $O(n \log n)$ segment fragments in all the cups, it takes $O(n \log^2 n)$ time and $O(n \log n)$ space to build the cup data structures and then to delete the segments. Shooting $\vec{e}$ into a cup takes $O(\log n)$ time.

The shooting algorithm shoots into one cup in each group S_i. It uses the cup endpoints to decide which cups to shoot into. It puts the cup endpoints from all the groups into one left-to-right list (resolving x-coordinate ties according to the order of the cups), then searches rightward in the list from the x-coordinate of p

to find a cup endpoint that lies above $\vec{e}$. Let z be the first such endpoint; the shooting algorithm shoots into the cup in each group S_i whose slab contains z.

Lemma 3.2. *Let z be the first cup endpoint in L that is right of p and above $\vec{e}$. Then $\vec{e}$ leaves the envelope of $\vec{e} \cup S$ in a cup whose slab contains z.*

Proof. By Lemma 3.1, $\vec{e}$ leaves the upper envelope of at least one group S_i in a cup whose slab contains z. Every cup strictly to the left of z lies below $\vec{e}$, since both its endpoints belong to L; any slab whose cup intersects $\vec{e}$ either contains z or lies to its right. □

The algorithm for finding the point z uses hull trees [11] once again. That data structure does not support insertions, but the updating of cup endpoints requires them. The algorithm resolves this problem by storing the endpoints of all $O(n \log n)$ canonical fragments in the hull tree—these are the present and future cup endpoints. This strategy works because the uppermost fragment endpoint at any slab boundary is also a cup endpoint.[1]

The hull tree is a complete binary tree that stores a set of points in its leaves, sorted in left-to-right order. (In our case the order is not strict.) Each subtree represents the convex hull of the points stored in its leaves; in particular, each node stores the upper common tangent of the convex hulls of its left and right subtrees. Each node also has an associated x-value, either the x-coordinate of its data point, in the case of a leaf, or the x-coordinate of a vertical line separating its left and right subtrees' data points, in the case of an internal node. (See Fig. 2.)

The search algorithm begins by walking down the hull tree to find the rightmost leaf to the left of p or with the same x-coordinate. Let T be the set of nodes to the right of the search path whose parents are on the path. The nodes of T are the roots of $O(\log n)$ disjoint subtrees whose union contains all points strictly to the right of p. The algorithm searches these $O(\log n)$ trees in left-to-right order, looking for a point that lies above $\vec{e}$.

To search for the leftmost point that lies above $\vec{e}$ in a subtree, the algorithm compares the position of the ray with that of the tangent edge stored at the root

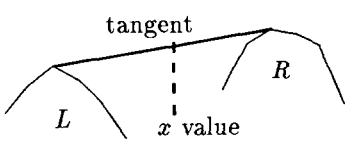


Fig. 2. The tangent stored at a hull tree node.

[1] The hull tree actually can be made to support replacement of a point by a lower point at the same x-coordinate, but for ease of exposition we ignore this possibility. The extra $\Theta(n \log n)$ space required to store all the fragment endpoints does not affect the asymptotic space complexity of the algorithm.

of the subtree, then recursively searches in the children of that node. The search may enter both children, but the following lemma shows that the entire search algorithm nonetheless takes only $O(\log^2 n)$ time altogether.

Lemma 3.3. *Using the hull tree data structure for a set of* $O(n \log n)$ *points, one can find the leftmost point that lies above a rightward-pointing ray* $\vec{e}$ *in* $O(\log^2 n)$ *time.*

Proof. To prove the time bound, we need to distinguish between successful and unsuccessful searches. We show that an unsuccessful search in a tree of height h takes $O(h)$ time, and that a successful search takes $O(h^2)$ time. The search for a point above $\vec{e}$ and right of its origin looks at $O(\log n)$ trees in left-to-right order. If any search finds a point above $\vec{e}$, the search stops. Hence there are $O(\log n)$ unsuccessful searches and at most one successful search, for a total cost of $O(\log^2 n)$.

When the search algorithm visits a node v with tangent edge t, there are four possible configurations of t and $\vec{e}$, shown in Fig. 3. In the first two cases, $\vec{e}$ passes above t, and the search proceeds into exactly one of the subtrees of v, the left one in case (a) and the right one in case (b). If $\vec{e}$ passes below the left endpoint of t (case (c)), the search proceeds in the left subtree—the left endpoint of t is evidence that the search will succeed. Case (d), in which $\vec{e}$ passes above the left endpoint of t but below the right endpoint, is the hard case. There may or may not be a point above $\vec{e}$ in the left subtree, but there certainly is one in the right subtree. The algorithm searches in the left subtree, and if that search fails, it then searches in the right subtree.

Let $f(h)$ be the cost of a successful search on a tree of height h, and let $g(h)$ be the cost of an unsuccessful search. The base values are $f(1) = g(1) = O(1)$. In an unsuccessful search, the algorithm sees only cases (a) and (b), in which the search proceeds into only one subtree. Hence the cost of the search is

$$g(h) = g(h-1) + O(1) = O(h).$$

In cases (c) and (d) the search will end successfully. Case (c) searches only one subtree, but case (d) may search two. The more expensive case occurs when both

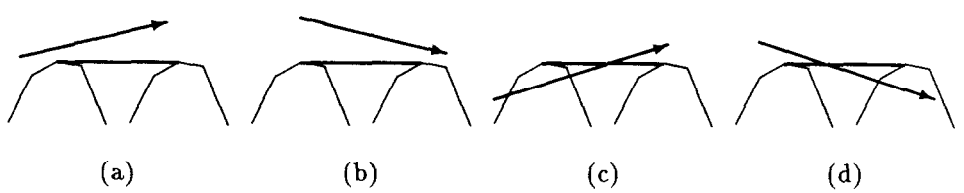


Fig. 3. Possible configurations of $\vec{e}$ and t.

subtrees are searched in case (d):

$$f(h) = g(h-1) + f(h-1) + O(1) = f(h-1) + O(h) = O(h^2).$$

This completes the proof. □

By using dynamic fractional cascading [13] and the ray-shooting algorithm of Chazelle and Guibas [5], we could reduce the query time of Lemma 3.3 to $O(\log n \log\log n)$. Unfortunately, that approach would increase the cost of maintaining the data structure to $O(n \log^2 n \log\log n)$ altogether, so the overall effect of the change would be negative.

The discussion so far shows that the shooting algorithm takes $O(\log^2 n)$ time to find the point q where a segment e leaves the upper envelope of S. If e leaves because it intersects some segment, then that segment is the next envelope segment. However, if e leaves because it goes beneath the left endpoint of some other segment (possibly more than one), or because its right endpoint is encountered, then the algorithm must find the uppermost segment immediately to the right of q. This can be done using at most three more shooting queries with vertical and horizontal rays. The same operation is used to find the leftmost segment of the upper envelope at the beginning of each iteration.

The algorithm described in this section finds an upper envelope of complexity k in time $O(k \log^2 n)$. The cost of deleting a segment that appears on the upper envelope is $O(\log^2 n)$, and the overhead of the data structure is $O(n \log^2 n)$ time and $O(n \log n)$ space.[2] This establishes the following theorem.

Theorem 3.4. *The upper envelope layers of a set of n possibly-intersecting line segments can be found in $O(n\alpha(n)\log^2 n)$ time and $O(n \log n)$ space.*

Remark. A simplification of this algorithm can compute the upper envelope layers for disjoint segments in $O(n \log n)$ time and space.

4. Parallel complexity

This section shows that the envelope layers problem is P-complete even for disjoint segments, and therefore intractable in parallel unless P = NC [7, 14]. The core of the proof is a reduction from the monotone circuit value problem (which is P-complete) to the envelope layers problem. Given a monotone circuit and an assignment of input values, we produce a segment arrangement that simulates the circuit under the assignment of input values. The outputs of the circuit are encoded by the layer number of certain output segments.

[2] Initialization time can be reduced to $O(n \log n)$, but this does not affect the asymptotic complexity of the algorithm.

Our proof uses a standard version of the monotone circuit value problem [7]. The circuit is represented by a directed acyclic graph (DAG). The nodes of the graph are *inputs* if they have indegree 0, *outputs* if they have indegree 1 and outdegree 0, and *gates* if they have indegree 2. Gates may have arbitrary outdegrees. The gates are of two types, ANDs and ORs. Edges of the graph play the rôle of wires connecting the circuit components. An instance of the problem specifies a circuit and an assignment of the values TRUE and FALSE to the inputs. We assume that the nodes of the DAG are topologically sorted, with all inputs at the beginning of the ordering and all outputs at the end. (Topological sorting is in NC^2.)

Given an instance of the monotone circuit value problem of size n, we construct an arrangement of $O(n^2)$ disjoint horizontal segments and an assignment of circuit outputs to segments such that the layer depth of an output segment is congruent to zero modulo 3 if and only if the corresponding circuit output is FALSE. We first describe the form of the arrangement, then focus on the details of segment placement.

The segment arrangement we build is laid out on a grid. The grid has one column for each edge of the DAG and one row for each gate, ordered from top to bottom according to the node ordering. (For convenience, we draw input columns to the left of output columns, but this is not essential.) The rows of the grid are numbered from 0 to m, where m is the number of gates; row 0 is used to set depths for the gates in rows 1 through m. All segments are horizontal and have endpoints on the column boundaries. There is no separation between adjacent columns. (We want the gate segments in row k to have depth $3k \pm 1$. If there were gaps between columns, and a gate segment had no other segments above it in some gap, then its depth would be forced to 1. We could separate the columns if we placed $3m + 1$ short segments above each gap, but is simpler to leave the columns adjacent. The restriction of the endpoints to lie on $O(n)$ vertical lines could be regarded as a degeneracy. We will see below how to perturb the segments so that no two endpoints lie on the same vertical line.) Each cell of the grid has one of two structures, as shown in Fig. 4.

Before we describe the segments, let us review the rules that govern them. Recall that the depth of a segment s is one more than the minimum depth, over all columns spanned by the segment, of the maximum depth segment above s in

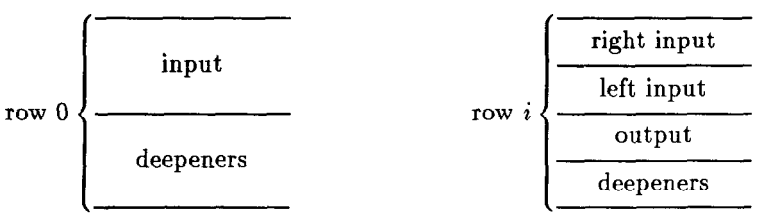


Fig. 4. The two cell configurations.

the column. If we know that s spans a column that contains d segments above s, then the depth of s is at most $d+1$, and no segment above s with depth at least $d+1$ can affect the depth of s. Conversely, in any column in which some segment s' above s has greater depth than s, we know that s cannot be the maximum depth segment in that column below s.

Each row of the segment arrangement simulates one gate of the circuit (except row 0, which simulates the circuit inputs). In each column that is not an input or an output of gate k, there is at least one segment above row k with depth greater than $3k$. In the input and output columns, the depths of the segments in and above row k are chosen to force the output segment of gate k to be at depth $3k$ or $3k+1$, depending on the inputs and the gate being simulated. The output depth is $3k$ if the output of the gate is FALSE, and $3k+1$ if the output is TRUE. The segments in the gate gadget (the segments in row k that extend between input and output columns) have depth at most $3k+1$. Because every column not associated with gate k contains a segment above row k with depth at least $3k+1$, the gate gadget is unaffected by those columns, and vice versa.

Each column of the segment arrangement acts as a wire. The signal on the wire is encoded by the depth of the deepest segment above the row where the signal is used. Just below the row in which the value of the signal traveling on the wire is set, the arrangement contains a group of *deepener* segments. This group spans exactly one column and contains the right number of segments to increase the depth in the column to match the level of the gate or output where the signal is used. Once the signal has been used as a gate input, another group of deepener segments in the gate row increases the depth in the column to at least $3m+1$, preventing this column from affecting lower gates.

The segment arrangement has three kinds of segments: input segments, deepeners, and gate gadgets. We consider each of these in turn, then prove the correspondence between the segment arrangement and the circuit.

The input slot of row 0 is used only in columns that carry circuit inputs. If the input associated with a particular column is TRUE, then we put a segment spanning the column in the input slot; otherwise, we leave the slot empty.

Deepener segments set the levels of gates and propagate signals from one row of the grid to another. If a column carries the output of gate k, we place $3k$ segments in the deepener slot of row 0 of that column. If a column carries a signal from row j to the gate in row k, with $0 \leq j < k$, we place $3(k-j)-2$ deepeners in row j of that column. Consequently, the deepest segment above row k in this column has depth $3k-2$ or $3k-1$, depending on whether the signal propagating in the column is FALSE or TRUE. If the output of gate k is connected to a circuit output via some column, then we place $3(m+1-k)$ segments in the deepener slot of row k of that column. We also place $3(m+1-k)$ deepeners in row k of the input columns for gate k. Refer to Fig. 5 for examples of deepener placement.

The gate gadget for gate k lies entirely in row k. If the inputs of gate k are

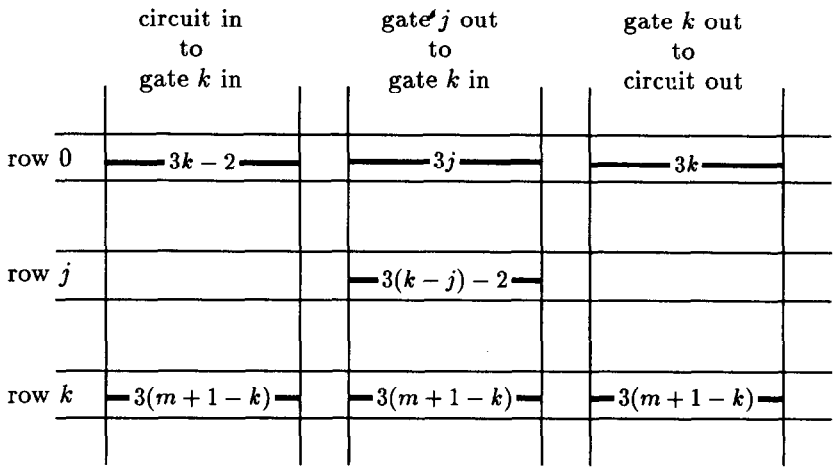


Fig. 5. Placing deepeners for two inputs and one output of gate k. The notation '—d—' represents d deepeners.

columns i and j, $i<j$, then column i contains a segment in the 'left input' slot of row k; column j contains a segment in the 'right input' slot; a segment in the output slot reaches from column i or j to the rightmost column for the output of gate k. The exact configuration of the gadget depends on whether gate k is an AND or an OR, as shown in Fig. 6.

The following lemmas show that the segment arrangement solves the given instance of the monotone circuit value problem. (The assumptions made in Lemma 4.1 are established by Lemma 4.2.) We use the notation depth(i, k) to refer to the depth of the maximum depth segment in column i above row k. We adopt the convention that if the depth of a signal-carrying segment may be either

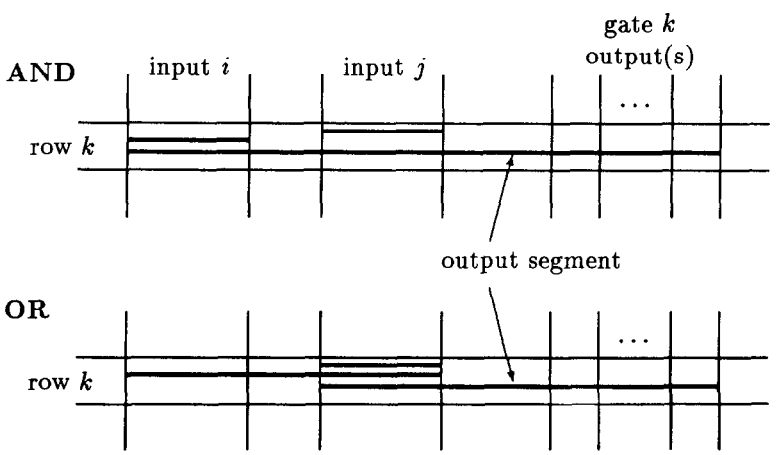


Fig. 6. AND and OR gate gadgets.

d or $d+1$, the lower value means the signal is FALSE and the higher means the signal is TRUE. We call this the *truth convention.*

Lemma 4.1. *Let $\bar{C}_k$ be the set of columns that do not carry inputs or outputs of gate k. Suppose that* $\text{depth}(i, k) > 3k$ *for all* $i \in \bar{C}_k$, $\text{depth}(i, k) \in \{3k-2, 3k-1\}$ *for each input i of gate k, and* $\text{depth}(i, k) = 3k$ *for each output i of gate k. Then the output segment of gate k has depth $3k$ or $3k+1$, satisfying the truth convention as a function of the inputs. The segments in row k do not affect segment depths in columns of $\bar{C}_k$.*

Proof. Because $\text{depth}(i, k) = 3k$ in each output column i, the output segment of gate k has depth at most $3k+1$. (Refer to Fig. 6.) Since $\text{depth}(i, k) > 3k$ for all $i \in \bar{C}_k$, none of the columns in $\bar{C}_k$ affects the depth of the output segment, and vice versa.

We first consider the AND gate gadget. The segments just above the output segment in the two input columns have depths in the range $\{3k-1, 3k\}$. The depth of the output segment is one more than the minimum of these segments' depths, and so the gate computes the AND function.

In the OR gate gadget, all the computation occurs in the input column on the right. The upper segment of the gadget (the one contained in the right column) has depth $3k-1$ or $3k$. The lesser of these depths would limit the left segment (the one that spans both input columns) to have depth at most $3k$, but this does not interfere with the value set by the left input column, which is $3k-1$ or $3k$. The depth of the output segment is one more than the maximum of the upper and left segments' depths, and so the gate computes the OR function. □

Lemma 4.2. *Let $\bar{C}_k$ be the set of columns that do not carry inputs or outputs of gate k. Then* $\text{depth}(i, k) > 3k$ *for all* $i \in \bar{C}_k$, $\text{depth}(i, k) \in \{3k-2, 3k-1\}$ *for each input i of gate k, and* $\text{depth}(i, k) = 3k$ *for each output i of gate k.*

Proof. The proof is by induction on k. The inputs to gate 1 must be inputs to the circuit, so the input columns have at most one segment in the input slot of row 0 and exactly $(3-2) = 1$ segment in the deepener slot. Thus $\text{depth}(i, 1) \in \{1, 2\}$ for each input i of gate 1. Clearly, $\text{depth}(i, 1) = 3$ for each output i. Each column i in $\bar{C}_k$ is either an input to some gate $j > 1$ (so $\text{depth}(i, 1) \geq 3j-2 > 3$) or the top of an output column for some gate $j > 1$ (so $\text{depth}(i, 1) = 3j > 3$).

Now consider $k > 1$. Each column i that carries an input to the circuit down to gate k has $\text{depth}(i, 1) = 3k-2+\{0 \text{ or } 1\}$, which is $3k-2$ or $3k-1$. By Lemma 4.1 the output segment of any gate $j < k$ has depth $3j$ or $3j+1$. Hence if column i carries the output of gate j to an input of gate k, the $3(k-j)-2$ deepeners below the output segment in row j ensure that $\text{depth}(i, j+1) \in \{3k-2, 3k-1\}$. If i is an output column of gate k, the deepeners at the top of the column set

depth$(i, 1) = 3k$. Lemma 4.1 guarantees that no segment above row k will change these values set in rows 0 and j, and so depth(i, k) is correctly set in all these cases.

Each column i in $\bar{C}_k$ is either (a) an input to some gate $j > k$ (so depth$(i, k) \geq 3j - 2 > 3k$), (b) the top of an output column for some gate $j > k$ (so depth$(i, k) = 3j > 3k$), (c) a circuit output from gate $j < k$ (so depth$(i, k) \geq 3m + 3 > 3k$), or (d) the bottom of an input column for gate $j < k$ (so depth$(i, k) \geq (3j - 1) + 3(m + 1 - j) > 3k$). Cases (a), (c), and (d) depend on Lemma 4.1, and case (d) also depends on the gate gadgets of Fig. 6. □

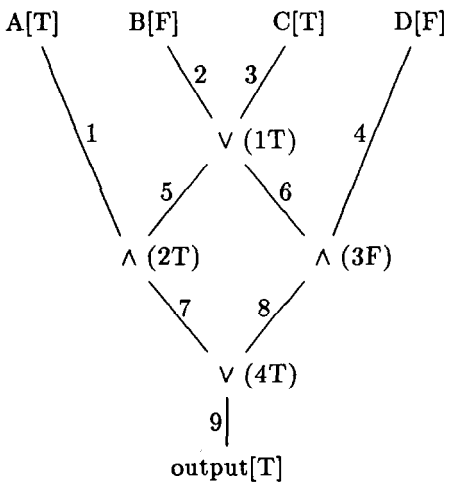
The depths of certain segments encode the outputs of the circuit. If column i carries a gate output to a circuit output, then depth$(i, m + 1)$ is either $3m + 3$ or $3m + 4$; this is the depth of the deepest segment fully contained in column i, namely a deepener in the gate row. This depth encodes the value of the circuit output according to the truth convention.

Theorem 4.3. *The arrangement of segments described above can be built in NC and simulates the monotone circuit on the given inputs. It follows that the envelope layers problem is P-complete.*

Proof. By induction and by Lemmas 4.1 and 4.2, each gate gadget receives inputs that encode the values of the corresponding gate inputs in the circuit, correctly simulates the gate it represents, and propagates its output onward. Gate k does not affect depths in any columns other than its input and output columns. The segment depths of inputs, gate outputs, and circuit outputs satisfy the truth convention.

The reduction from a topologically sorted monotone circuit to an arrangement of segments can be computed in LOGSPACE, and hence in NC, since each gate or wire maps to a gadget in the arrangement that can be specified at a high level using only $O(\log n)$ bits. The actual position of each segment takes $O(\log n)$ bits to specify: each row needs space for $O(1)$ gate segments and $O(n)$ deepener segments, and so the segments can be placed at integer y-coordinates with magnitude $O(n^2)$. The x-coordinates are integers with magnitude $O(n)$. □

If we want to avoid segments whose endpoints lie on the same vertical line, we can achieve this by lengthening each segment slightly. We order the segments from top to bottom, then lengthen the lower segments more than the upper ones. (For example, we could lengthen a segment with y-coordinate Y by $\varepsilon(Y_{\max} - Y)$ at both ends, where $Y_{\max}$ is the maximum y-coordinate of any segment.) Lengthening segments does not decrease the depth of any segment, and if we choose ε small enough (less than the column width divided by $2(Y_{\max} - Y_{\min})$), no depth will increase, either: each column will have a stripe down the middle that looks and functions just like the unmodified column. If we lengthen our segments



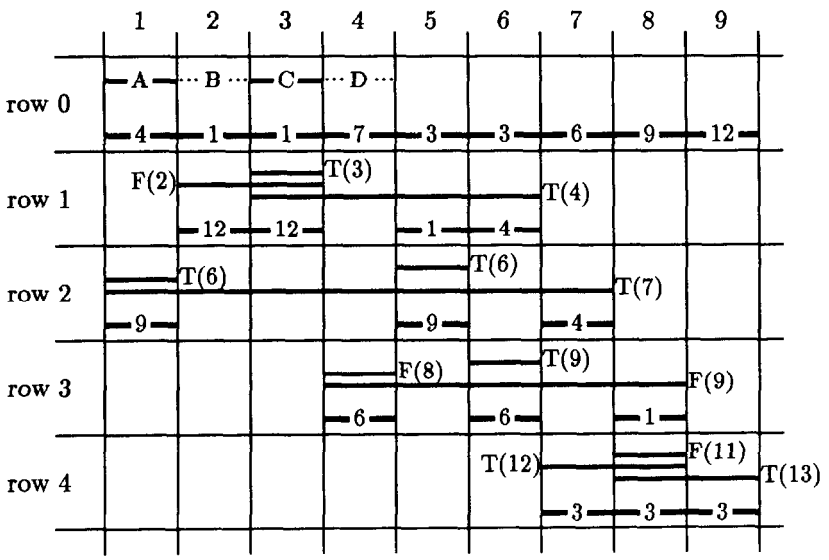


Fig. 7. A circuit for the formula $(A \wedge (B \vee C)) \vee ((B \vee C) \wedge D)$ and its segment arrangment. Each input to the circuit is assigned a truth value, and each gate i is labeled with i and the gate's output value. In the segment arrangement, TRUE inputs are solid segments, and FALSE inputs are missing (dotted) segments. The notation '—d—' represents d deepeners. Each segment of a gate gadget is labeled with its truth value and its depth.

in this way, then the x-coordinates of the segment endpoints have magnitude $O(n^3)$.

For an example of a segment arrangement simulating a circuit, see Fig. 7.

5. Open problems

Several problems remain unresolved. The first is that of improving the sequential algorithm for computing envelope layers. The lower bound for this problem is just $\Omega(n \log n)$, and an $O(n\alpha(n)\log n)$ algorithm does not seem out of reach.

The convex layers problem, which inspired this research into envelope layers, is still a major open problem in parallel computational geometry. Does it belong to NC, is it P-complete, or does it lie somewhere in between?

We have shown that the envelope layers problem is P-complete even for disjoint, horizontal segments. To understand the source of this complexity, we can consider a specialization suggested by Goodrich, in which all the segments are horizontal and have constant length. Similarly, we can consider segments with bounded length ratios. Are the resulting problems still P-complete? Our proof does not apply in these cases.

Acknowledgments

I would like to thank Mike Goodrich for proposing the envelope layers problem and for helpful discussions, Richard Anderson for clarifying the subject of P-completeness, and DIMACS, for sponsoring the workshop where this research began.

References

[1] P.K. Agarwal, Ray shooting and other applications of spanning trees with low stabbing number, in: Proceedings of the 5th ACM Symposium on Computational Geometry (1989) 315–325.

[2] Ta. Asano, Te. Asano, L. Guibas, J. Hershberger and H. Imai, Visibility of disjoint polygons, Algorithmica 1 (1) (1986) 49–63.

[3] M. Atallah, P. Callahan and M. Goodrich, P-complete geometric problems, in: Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures (1990).

[4] B. Chazelle, On the convex layers of a planar set, IEEE Trans. Inform. Theory, IT-31 (4) (1985) 509–517.

[5] B. Chazelle and L. Guibas, Visibility and intersection problems in plane geometry, in: Proceedings of the ACM Symposium on Computational Geometry (1985) 135–146.

[6] H. Edelsbrunner, Algorithms in Combinatorial Geometry, EATCS Monographs on Theoretical Computer Science, Vol. 10 (Springer, Berlin, 1987).

[7] R. Greenlaw, H.J. Hoover and W.L. Ruzzo, A compendium of problems complete for P, 1989, manuscript.

[8] L.J. Guibas and F.F. Yao, On translating a set of rectangles, in: Proceedings of the 12th ACM Symposium on Theory of Computing (1980) 154–160.
[9] S. Hart and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. Combinatorica 6 (1986) 151–177.
[10] J. Hershberger, Finding the upper envelope of n line segments in $O(n \log n)$ time, Inform. Process. Lett. 33 (1989) 169–174.
[11] J. Hershberger and S. Suri, Applications of a semi-dynamic convex hull algorithm, in: Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory. (Springer, Berlin, 1990) 380–392.
[12] R.E. Ladner, The circuit value problem is log space complete for P, SIGACT News 7 (1) (1975) 18–20.
[13] K. Mehlhorn and S. Näher, Dynamic fractional cascading, Algorithmica 5 (1990) 215–241.
[14] I. Parberry, Parallel Complexity Theory, Research Notes in Theoretical Computer Science (Pitman, London, 1987) and (Wiley, New York, 1987).
[15] F.P. Preparata and M.I. Shamos, Computational Geometry (Springer, Berlin, 1985).
[16] A. Wiernik and M. Sharir, Planar realization of nonlinear Davenport-Schinzel sequences by segments, Discrete Comput. Geom. 3 (1988) 15–47.